BGET -- Memory Allocator

by John Walker
http://www.fourmilab.ch/

BGET is a comprehensive memory allocation package which is easily configured to the needs of an application. BGET is efficient in both the time needed to allocate and release buffers and in the memory overhead required for buffer pool management. It automatically consolidates contiguous space to minimise fragmentation. BGET is configured by compile-time definitions, Major options include:

* A built-in test program to exercise BGET and demonstrate how the various functions are used.

* Allocation by either the "first fit" or "best fit" method.

* Wiping buffers at release time to catch code which references previously released storage.

* Built-in routines to dump individual buffers or the entire buffer pool.

* Retrieval of allocation and pool size statistics.

* Quantisation of buffer sizes to a power of two to satisfy hardware alignment constraints.

* Automatic pool compaction, growth, and shrinkage by means of call-backs to user defined functions.

Applications of BGET can range from storage management in ROM-based embedded programs to providing the framework upon which a multitasking system incorporating garbage collection is constructed. BGET

incorporates extensive internal consistency checking using the
<assert.h> mechanism; all these checks can be turned off by compiling with NDEBUG defined, yielding a version of BGET with minimal size and
maximum speed.

The basic algorithm underlying BGET has withstood the test of time; more than 25 years have passed since the first implementation of this code. And yet, it is substantially more efficient than the native allocation schemes of many operating systems: the Macintosh and Microsoft Windows to name two, on which programs have obtained substantial speed-ups by
layering BGET as an application level memory manager atop the underlying system's.

BGET has been implemented on the largest mainframes and the lowest of
microprocessors. It has served as the core for multitasking operating
systems, multi-thread applications, embedded software in data network
switching processors, and a host of C programs. And while it has
accreted flexibility and additional options over the years, it remains fast, memory efficient, portable, and easy to integrate into your
program.

BGET IMPLEMENTATION ASSUMPTIONS
===============================

BGET is written in as portable a dialect of C as possible. The only
fundamental assumption about the underlying hardware architecture is
that memory is allocated is a linear array which can be addressed as a
vector of C "char" objects. On segmented address space architectures, this generally means that BGET should be used to allocate storage within a single segment (although some compilers simulate linear address spaces on segmented architectures). On segmented architectures, then, BGET

buffer pools may not be larger than a segment, but since BGET allows any
number of separate buffer pools, there is no limit on the total storage
which can be managed, only on the largest individual object which can be
allocated. Machines with a linear address architecture, such as the
VAX, 680x0, Sparc, MIPS, or the Intel 80386 and above in native mode,
may use BGET without restriction.

GETTING STARTED WITH BGET

Although BGET can be configured in a multitude of fashions, there are
three basic ways of working with BGET. The functions mentioned below
are documented in the following section. Please excuse the forward
references which are made in the interest of providing a roadmap to
guide you to the BGET functions you're likely to need.

Embedded Applications

Embedded applications typically have a fixed area of memory dedicated to
buffer allocation (often in a separate RAM address space distinct from
the ROM that contains the executable code). To use BGET in such an
environment, simply call bpool() with the start address and length of
the buffer pool area in RAM, then allocate buffers with bget() and
release them with brel(). Embedded applications with very limited RAM
but abundant CPU speed may benefit by configuring BGET for BestFit
allocation (which is usually not worth it in other environments).

Malloc() Emulation

If the C library malloc() function is too slow, not present in your
development environment (for example, an a native Windows or Macintosh
program), or otherwise unsuitable, you can replace it with
BGET.
Initially define a buffer pool of an appropriate size with
bpool()--usually obtained by making a call to the operating system's
low-level memory allocator. Then allocate buffers with bget(), bgetz(),
and bgetr() (the last two permit the allocation of buffers initialised
to zero and [inefficient] re-allocation of existing buffers for
compatibility with C library functions). Release buffers by calling
brel(). If a buffer allocation request fails, obtain more storage from
the underlying operating system, add it to the buffer pool by another
call to bpool(), and continue execution.

Automatic Storage Management

You can use BGET as your application's native memory manager and
implement automatic storage pool expansion, contraction, and optionally
application-specific memory compaction by compiling BGET with the BECtl
variable defined, then calling bectl() and supplying functions for
storage compaction, acquisition, and release, as well as a standard pool
expansion increment. All of these functions are optional (although it
doesn't make much sense to provide a release function without an
acquisition function, does it?). Once the call-back functions have been
defined with bectl(), you simply use bget() and brel() to allocate and
release storage as before. You can supply an initial buffer pool with
bpool() or rely on automatic allocation to acquire the entire pool.
When a call on bget() cannot be satisfied, BGET first checks if a

compaction function has been supplied. If so, it is called (with the
space required to satisfy the allocation request and a sequence number
to allow the compaction routine to be called successively without
looping). If the compaction function is able to free any storage (it
needn't know whether the storage it freed was adequate) it should return
a nonzero value, whereupon BGET will retry the allocation request and,
if it fails again, call the compaction function again with the
next-higher sequence number.

If the compaction function returns zero, indicating failure to free
space, or no compaction function is defined, BGET next tests whether a
non-NULL allocation function was supplied to bectl(). If so, that
function is called with an argument indicating how many bytes of
additional space are required. This will be the standard pool expansion
increment supplied in the call to bectl() unless the original bget()
call requested a buffer larger than this; buffers larger than the
standard pool block can be managed "off the books" by BGET in this mode.
If the allocation function succeeds in obtaining the storage, it returns
a pointer to the new block and BGET expands the buffer pool; if it
fails, the allocation request fails and returns NULL to the caller. If
a non-NULL release function is supplied, expansion blocks which become
totally empty are released to the global free pool by passing their
addresses to the release function.

Equipped with appropriate allocation, release, and compaction functions,
BGET can be used as part of very sophisticated memory management
strategies, including garbage collection. (Note, however, that BGET is
not a garbage collector by itself, and that developing such a system
requires much additional logic and careful design of the application's
memory allocation strategy.)

BGET FUNCTION DESCRIPTIONS
==========================

Functions implemented by BGET (some are enabled by certain of the optional settings below):

```
void bpool(void *buffer, bufsize len);
```

Create a buffer pool of <len> bytes, using the storage starting at <buffer>. You can call bpool() subsequently to contribute additional storage to the overall buffer pool.

```
void *bget(bufsize size);
```

Allocate a buffer of <size> bytes. The address of the buffer is returned, or NULL if insufficient memory was available to allocate the buffer.

```
void *bgetz(bufsize size);
```

Allocate a buffer of <size> bytes and clear it to all zeroes. The address of the buffer is returned, or NULL if insufficient memory was available to allocate the buffer.

```
void *bgetr(void *buffer, bufsize newsize);
```

Reallocate a buffer previously allocated by bget(), changing its size to <newsize> and preserving all existing data. NULL is returned if insufficient memory is available to reallocate the buffer, in which case the original buffer remains intact.

```
void brel(void *buf);
```

Return the buffer <buf>, previously allocated by bget(), to the free space pool.

```
        void bectl(int (*compact)(bufsize sizereq, int sequence),
                   void *(*acquire)(bufsize size),
                   void (*release)(void *buf),
                   bufsize pool_incr);
```

Expansion control: specify functions through which the package may compact storage (or take other appropriate action) when an allocation request fails, and optionally automatically acquire storage for expansion blocks when necessary, and release such blocks when they become empty. If <compact> is non-NULL, whenever a buffer allocation request fails, the <compact> function will be called with arguments specifying the number of bytes (total buffer size, including header overhead) required to satisfy the allocation request, and a sequence number indicating the number of consecutive calls on <compact> attempting to satisfy this allocation request. The sequence number is 1 for the first call on <compact> for a given allocation request, and increments on subsequent calls, permitting the <compact> function to take increasingly dire measures in an attempt to free up storage. If the <compact> function returns a nonzero value, the allocation attempt is re-tried. If <compact> returns 0 (as it must if it isn't able to release any space or add storage to the buffer pool), the allocation request fails, which can trigger automatic pool expansion if the <acquire> argument is non-NULL. At the time the <compact> function is called, the state of the buffer allocator is identical to that at the moment the allocation request was made; consequently, the <compact> function may call brel(), bpool(), bstats(), and/or directly manipulate the buffer pool in any manner which would be valid were the application in control. This does not, however, relieve the <compact> function of the need to ensure that whatever actions it takes do not change things underneath the application that made the allocation request. For example, a <compact> function that released a buffer in the process of being reallocated with bgetr() would lead to disaster. Implementing a safe and effective <compact> mechanism requires careful design of an application's memory architecture, and cannot generally be easily retrofitted into existing code.

If <acquire> is non-NULL, that function will be called whenever an allocation request fails. If the <acquire> function succeeds in allocating the requested space and returns a pointer to the new area, allocation will proceed using the expanded buffer pool. If <acquire> cannot obtain the requested space, it should return NULL and the entire allocation process will fail. <pool_incr> specifies the normal expansion block size. Providing an <acquire> function will cause subsequent bget() requests for buffers too large to be managed in the linked-block scheme (in other words, larger than <pool_incr> minus the buffer overhead) to be satisfied directly by calls to the <acquire> function. Automatic release of empty pool blocks will occur only if all pool blocks in the system are the size given by <pool_incr>.

```
void bstats(bufsize *curalloc, bufsize *totfree,
            bufsize *maxfree, long *nget, long *nrel);
```

The amount of space currently allocated is stored into the variable

pointed to by <curalloc>. The total free space (sum of all free blocks in the pool) is stored into the variable pointed to by <totfree>, and
the size of the largest single block in the free space pool is stored into the variable pointed to by <maxfree>. The variables pointed to by
<nget> and <nrel> are filled, respectively, with the number of
successful (non-NULL return) bget() calls and the number of brel()
calls.

```
        void bstatse(bufsize *pool_incr, long *npool,
                     long *npget, long *nprel,
                     long *ndget, long *ndrel);
```

Extended statistics: The expansion block size will be stored into the
variable pointed to by <pool_incr>, or the negative thereof if automatic expansion block releases are disabled. The number of currently active
pool blocks will be stored into the variable pointed to by <npool>. The variables pointed to by <npget> and <nprel> will be filled with,
respectively, the number of expansion block acquisitions and releases
which have occurred. The variables pointed to by <ndget> and <ndrel> will be filled with the number of bget() and brel() calls, respectively, managed through blocks directly allocated by the acquisition and release functions.

```
        void bufdump(void *buf);
```

The buffer pointed to by <buf> is dumped on standard output.

```
        void bpoold(void *pool, int dumpalloc, int dumpfree);
```

All buffers in the buffer pool <pool>, previously initialised by a call on bpool(), are listed in ascending memory address order. If
<dumpalloc> is nonzero, the contents of allocated buffers are dumped; if
<dumpfree> is nonzero, the contents of free blocks are dumped.

```
        int bpoolv(void *pool);
```

The named buffer pool, previously initialised by a call on bpool(), is
validated for bad pointers, overwritten data, etc. If compiled with
NDEBUG not defined, any error generates an assertion failure. Otherwise 1
is returned if the pool is valid, 0 if an error is found.

BGET CONFIGURATION
==================

```
#define TestProg    20000  /* Generate built-in test program
                              if defined.  The value specifies
                              how many buffer allocation attempts
                              the test program should make. */

#define SizeQuant   4      /* Buffer allocation size quantum:
                              all buffers allocated are a
                              multiple of this size.  This
                              MUST be a power of two. */

#define BufDump     1      /* Define this symbol to enable the
                              bpoold() function which dumps the
                              buffers in a buffer pool. */

#define BufValid    1      /* Define this symbol to enable the
                              bpoolv() function for validating
                              a buffer pool. */

#define DumpData    1      /* Define this symbol to enable the
                              bufdump() function which allows
                              dumping the contents of an allocated
                              or free buffer. */

#define BufStats    1      /* Define this symbol to enable the
                              bstats() function which calculates
                              the total free space in the buffer
                              pool, the largest available
                              buffer, and the total space
                              currently allocated. */

#define FreeWipe    1      /* Wipe free buffers to a guaranteed
```

```
                                   pattern of garbage to trip up
                                   miscreants who attempt to use
                                   pointers into released buffers. */

#define BestFit      1          /* Use a best fit algorithm when
                                   searching for space for an
                                   allocation request.  This uses
                                   memory more efficiently, but
                                   allocation will be much slower. */

#define BECtl        1          /* Define this symbol to enable the
                                   bectl() function for automatic
                                   pool space control.  */
```